LOW COST WIRELESS SYSTEM USING COTS MODULE

By

Abeer Asif

Mobeen Tahir

Amna Kamran

Luqman Khalid

Submitted to the Faculty of Department of Electrical Engineering,

Military College of Signals National University of Sciences and Technology, Islamabad

in partial fulfillment for the requirements of a B.E Degree in

Telecom Engineering

June 2016

ABSTRACT

LOW COST WIRELESS SYSTEM USING COTS MODULE

While the security needs of Pakistan are on the rise, the purpose of this project is the design of a Low Cost Wireless System using COTS Module to meet the demanding needs of a soldier. The project is designed to meet specifications that were set to ensure that the Low cost wireless system would be able to operate in a battle zone. Low cost wireless systems are an essential part of modern warfare for better performance and co-ordination of the soldiers. The project accomplishes short-distance wireless communication using ZigBee Technology. Secure and efficient transmission of voice, without compromising on the voice quality is the main aim of project, which is achieved by compression using the MPEG-2.5 Audio Layer III. Analog voice signals received by microphone are converted to a stream of raw data by the built-in ADC which are then converted to mp3 compressed format and using XBee S2, sent over the air wirelessly to the destination where a reverse procedure is applied. This ensures fast and quality transfer of voice while providing the required security and efficiency required by the Pakistan Army.

CERTIFICATE

It is hereby certified that the contents and form of the project report entitled "**Low Cost Wireless System using COTS Module"** submitted by 1) Amna Kamran 2) Mobeen Tahir 3) Abeer Asif 4) Luqman Khalid have been found satisfactory as per the requirement of the B.E. Degree in Electrical (Telecom) Engineering.

Supervisor:

Lt.Col SaifUllah Khalid

Military College of Signals

National University of Sciences and Technology

_____

DECLARATION

We hereby declare that no content of work presented in this thesis has been submitted in support of another award of qualification or degree either in this institution or anywhere else.

DEDICATED TO

Allah Almighty,

Supervisor for his help

And our parents for their support

ACKNOWLEDGEMENTS

Nothing happens without the will of Allah Almighty. We thank Allah Almighty for giving us knowledge and strength to accomplish this task successfully.

We would like to thank our project supervisor, Lt. Col SaifUllah Khalid, without whose support and encouragement; it would not have been possible to complete this project.

We would also like to thank our colleagues for helping us in developing the project and appreciate the people who have willingly helped us with their abilities and given us their time.

Last but not the least; we are very thankful to our parents, who helped us in times of difficulty and hardship. Without their consistent support and encouragement, we could not have accomplished our targets successfully.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF TABLES

LIST OF ABBREVIATIONS

COTS: Commercial off the shelf

RF: Radio Frequency

WPC: Wireless Power Controller

PRR: Personal Role Radio

WLAN: Wireless Local Area Network

ADC: Analog to Digital Converter

API: Application Programming Interface

FCC: Federal Communications Commission

DTE: Data Terminal Equipment

DCE: Data Communications Equipment

IDE: Integrated Development Environment

OS: Operating System

OOP: Object Oriented Programming

PC: Personal Computer

IDLE: Integrated Development Environment (Python)

A/D: Analog to Digital Converter

D/A: Digital to Analog Converter

Av Converter: Audio Video Converter

FIFO: First In First Out

Chapter 1

INTRODUCTION

1.1 BACKGROUND OVERVIEW

We are living in one of the most dangerous eras of history, with terrorism, internal unrest and conflicts on the rise. Under these circumstances, the need for highly trained soldiers equipped with heavy ammunition and efficient technology is of the utmost importance. This requirement along with the motivation to make a secure wireless system led to the development and implementation of this project. Pakistan is a third world country with increasing demands and scarce resources. Getting aid and borrowing equipment from international markets is not as secure and reliable, thus the need to make our very own and secure wireless system gave birth to this project.

Low Cost Wireless System using COTS Module is a need of the soldiers in the era of modern warfare. It will effectively allow the soldiers to have better communication during an on-going operation without any interference from the harsh environment. It will enable section commanders to react quickly and efficiently to rapidly changing situations, including contact with the enemy, greatly increasing the effectiveness of infantry fire teams.

1.2 PROJECT DESCRIPTION AND SALIENT FEATURES

Low Cost Wireless System using COTS Module includes two modules; Controller Module and RF Module

The project aims at providing efficient transmission of voice from sender to receiver without compromising on the voice quality. This is achieved by compressing the voice before it sent over the transmitting RF Module serially. Our controller module is Linux

Based which allows a wide range of libraries to be used and minimize the work load. During the project, three different controller modules have been used which have been described in the Design and Development section. Only one RF Module has been used, which is Xbee S2. The reason for using Xbee is that it provides secure transmission because of its built-in four layer architecture that provides the much needed security.

Basic design function along with sequential methodology has been shown in the figures below

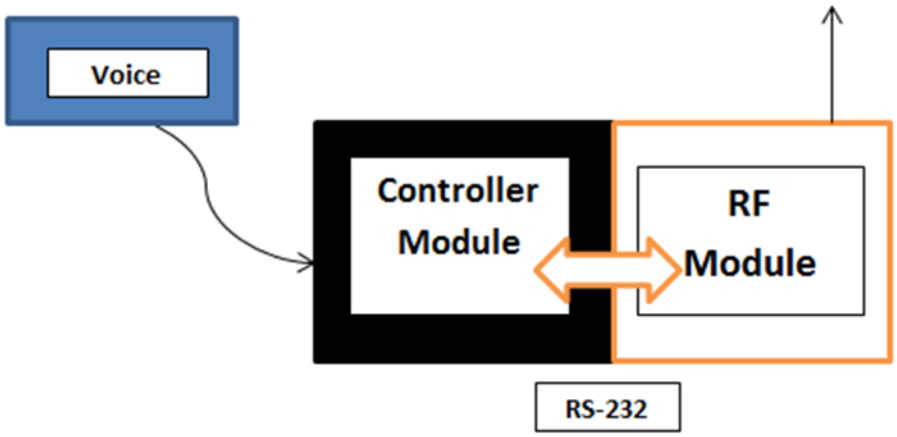


Figure 1-1: Basic Design Function

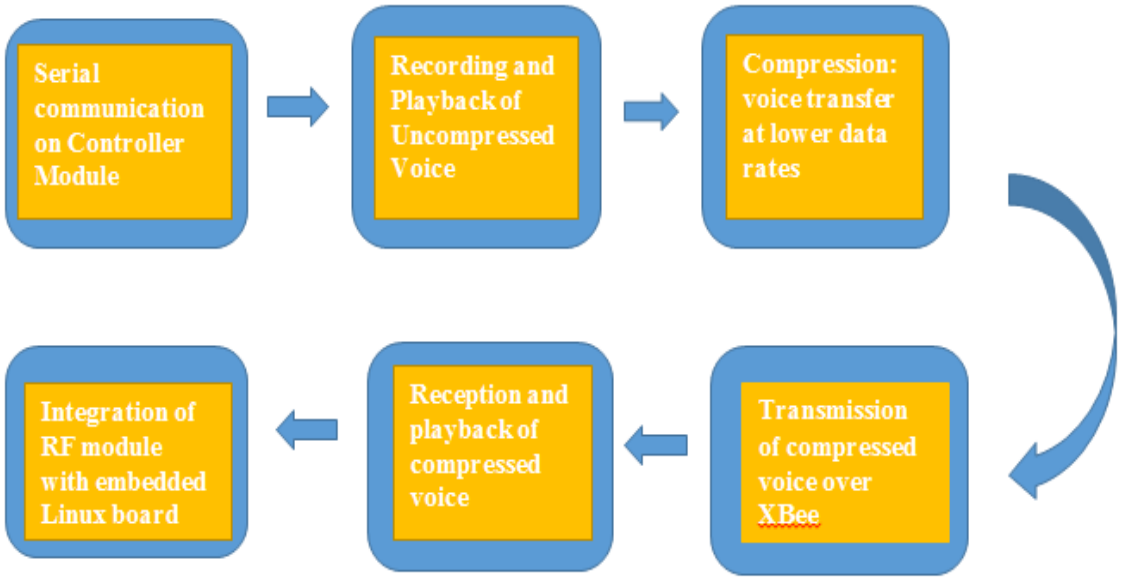


Figure 1-2: Sequential Methodology

Applications of low cost wireless systems are vast. There are different types of low cost wireless systems being used around the world. A few examples are given below:

- Low Cost Wireless High Water Detection System
- Low System Cost, Wireless Power Controller for WPC TX A6
- Low Cost Wireless Healthcare Monitoring System

However, the aim of this project is to develop a simplex transmitter receiver system capable of providing communication at lower data rates without compromise on voice quality and security. Similar applications have been developed at a much more advanced level and are being used by the British Army, Royal Marines, Royal Navy and the Royal

Air Force Regiment. It allows users to communicate over short distances. It is called a Personal Role Radio and is issued to every member of the infantry section.

1.3 SCOPE AND OBJECTIVES

Low Cost Wireless System using COTS Module uses simple and easy to use technology available. It makes use of commonly available hardware that is affordable and easy to configure. It can be used in almost every part of our lives and not just in the battle zone. The versatility of the project lies in its ability to be used in our day to day lives to tense hostage situations.

Scope of the work includes:

1) Literature review to evaluate the recent developments in this area
2) Serial communication on Ubuntu Pc
3) Voice compression and transmission
4) Implementation using RF Module and performance characterization on Embedded Board

The objectives of the project are:

1) to design a wireless communication system using COTS equipment
2) use commonly available RF hardware modules (Zigbee modules, RTL-SDR) and Embedded Linux board
3) develop firmware for interfacing the RF module and Baseband Processing module

1.4 DELIVERABLES OF THE PROJECT

A small, secure and low cost ZigBee based radio set prototype (small range), complete in its communication and electrical capabilities along with a prototype power system that will be used to demonstrate how clearly and efficiently the compressed voice is transmitted onto another ZigBee based radio set, thus helping the soldiers to communicate efficiently for war fare in today's world.

1.5 FINALIZED APPROACH

The project was successfully completed using the finalized approach mentioned herein. It involved using Ubuntu PC as the Controller Module which was programmed in C to record the audio and transmit to the receiving side. The transmitting end simultaneously recorded the voice and compressed it using MPEG compression while serially transmitting it to the Xbee S2 RF Module, which further transmitted the compressed voice over the air, over to the receiving end. At the receiving end, Raspberry Pi Kit 2 has been employed as the Controller Module connected to another Xbee S2 RF Module which receives the compressed voice, decompresses it and plays it back. Further details have been explained fully in Chapter 4 titled 'System Design and Development'.

Chapter 2

LITERATURE REVIEW

The purpose of the project “Low Cost Wireless System using COTS Module” is to design a low cost and a secure wireless system which can be developed using COTS module. It can be used by the Pakistan Army. Pakistan at the moment is incapable of making high level equipment on its own, however it is the need of the hour for the Pakistan Army to deploy radio sets to each and every soldier in the warfare and to successfully change the specifications of foreign equipment in order to make it suitable for our environment as the equipment that Pakistan buys from outside world cannot be trusted. There is always a chance of being watched/heard as the equipment is vulnerable to brute-force attacks which is a serious threat to security and integrity of our messages.

Following is a brief description of the relevant background study and other project details before we move on to the detailed design,

2.1 LANGUAGES

2.1.1 PYTHON

Python offers many choices for web development. Python's standard library supports many Internet protocols:

- HTML and XML
- JSON
- E-mail processing

This language is widely used in scientific and numeric computations. Python is often used as a support language in order to build control and management for software developers for maintenance, testing and in many other ways.

2.1.2 C LANGUAGE

C language is a building block for many other operating programming languages. Due to having a variety of powerful operators, easy syntax and data types, programs written in C language are efficient, fast and easy to understand. C language is known for its ability to extend itself. Various functions in C language are supported by the C library which makes it really helpful for us to add our own features to the C library. Presence of a large number of functions makes the programming quite simple. The modular structure in C language makes the testing, debugging and maintenance easier.

2.1.3 C++ LANGUAGE

C++ is an object oriented programming language.

C++ is a general purpose programming language better than ‘c’. General purpose means it is suitable for developed any type of software.

Final program was developed using C++, so we compare their advantages over each other:

C language	C++ language
No virtual functions are present	The concept of virtual functions is used in C++
In C, Polymorphism is not possible	The concept of polymorphism is used in C++. Polymorphism is the most important feature of OOP.
Operator overloading is not possible in C.	Operator overloading is one of the greatest features of C++
Mapping between Data and Function is difficult and complicated	Mapping between Data and Function can be used using "Objects"
C requires all the variables to be defined at the starting of a scope	C++ allows the declaration of variable anywhere in the scope i.e. at time of its first use

Table 2-1

2.2 DIGITAL SIGNAL PROCESSING

Digital signal processors are a kind of microcomputers whose hardware, software and sets of instruction are optimized for fast and high-speed numeric processing applications which are essential for processing digital data and to represent them as analog signals in real time. When acting as a digital filter, the DSP receives digital values based on

samples of a signal, better the sampling rate better the quality of signal. It calculates the results of a filter function operating on these values and provides resulting values that represent the filter output. It can also provide system control signals based on properties of these values. High-speed arithmetic and logical hardware is programmed to rapidly execute algorithms modelling the output function according to the parameters we set.

The basic process model:

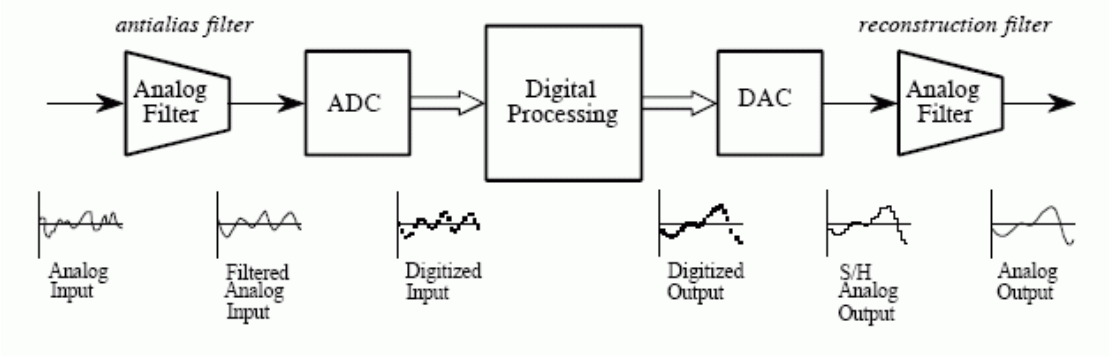


Figure 2-1: Filtering Process

2.2.1 SAMPLING

Sampling is the process of converting a continuous time signal to a discrete signal. A sample consists of a set of values at particular defined points in time or space. The reconstruction of signal after sampling at receiver end exhibits imperfection which is known as ‘aliasing’. Aliasing can be defined as distortion or simple overlapping of the signals as we might have more number of samples that are required. In the other case, the number of samples is simply small that might lead to distortion. In order to overcome this problem, we employ the famous **Nyquist Criteria** for sampling of voice in our project:

‘The sampling frequency should be at least twice the highest frequency contained in the signal’

The voice is being sampled at **8 kHz**, 4 kHz being the bandwidth for voice range.

2.2.2 QUANTIZATION

In mathematics and digital signal processing, quantization is the process of mapping a large set of input values to a (countable) smaller set (amplitude). The best practice in order to quantize a signal is to choose an amplitude value of signal at a particular point which is closest to the value of amplitude in original analogue signal.

2.2.3 COMPRESSION USING AVCONV

AVconv is a library that is being employed in our code to achieve successful accomplishment of compressing voice from a large size to an mP3 file which is of much smaller size. AVconv can switch between arbitrary sample rates in order to change file size. The transcoding process involved in changing the file size after the processing of the signals is as follows:

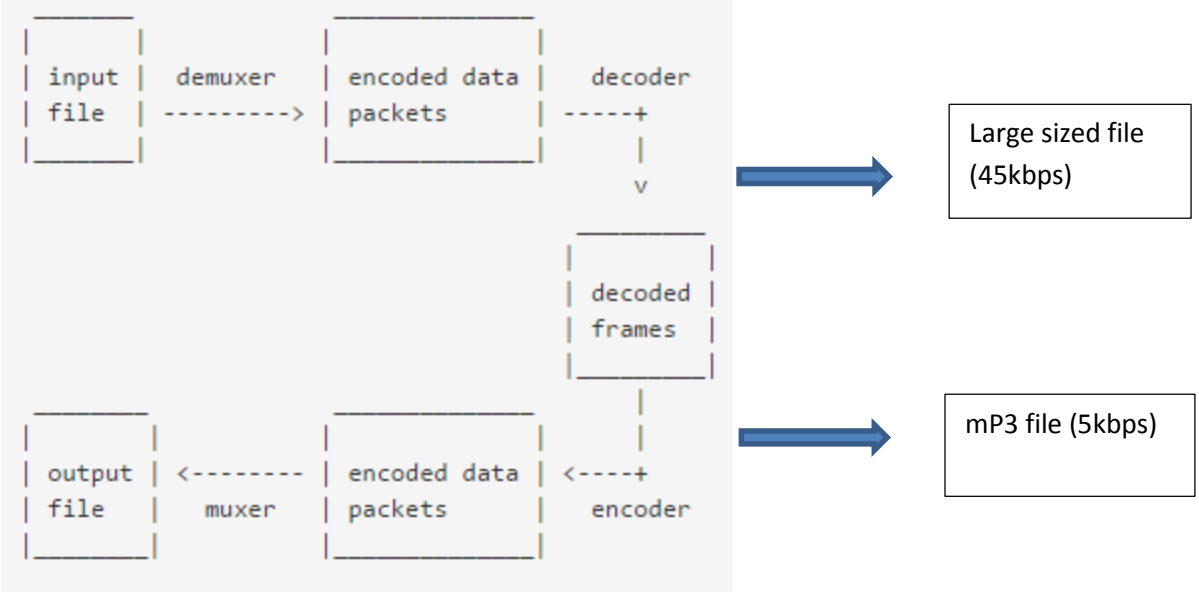


Figure 2-2: Av converter

2.3 WAVE PROPAGATION AND ANTENNA THEORY

Our project involves wireless transmission of the signals as ZigBee modules are being used for transmission and reception. It is important to have a basic knowledge of how signals are sent over the air-interface for which understanding of the above mentioned topic was necessary.

2.4 UNDERSTANDING SECURITY PROVISION IN THE PROJECT

The security and integrity of voice being transmitted is ensured by ZigBee module itself that has an in-built 128-bit AES encryption standard installed in the MAC layer. 128-bit encryption is a data/file encryption technique that uses a 128-bit key to encrypt and decrypt data or files. 128-bit encryption is one of the most secure encryption techniques used in modern encryption algorithms and technologies. It is considered to be logically unbreakable.

How it works:

128 bit refers to the length/size of key used in the process. It provides massive security as it would take thousands of years to compute and break the cipher e.g. it would take 2^128 combinations to be checked and break the encryption key which is simply out of reach of even the most powerful computers.

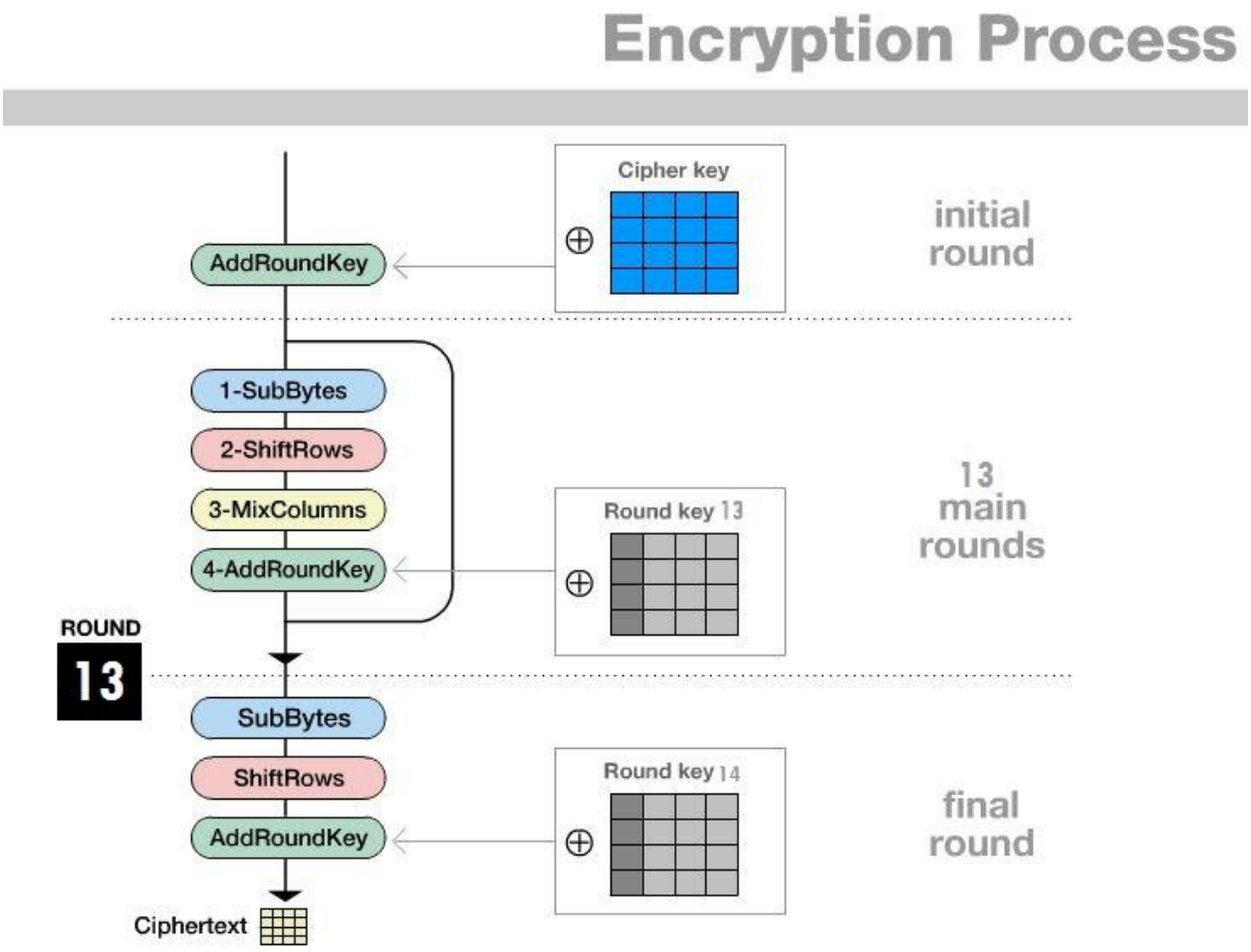


Figure 2-3: Encryption Process

Chapter 3

SYSTEM DESIGN AND DEVELOPMENT

3.1 TECHNICAL SPECIFICATIONS:

3.1.1 XBEE S2B

RF Module used for our project is Xbee S2. It provides short distance wireless communication, with a range of 400ft. It operates within the WLAN Bandwidth of 2.4 Ghz.

Series 2 improves on the power output and data protocol. These modules have allowed us to create complex mesh networks based on the XBee ZB ZigBee mesh firmware. These modules have also allowed a very reliable and simple communication between microcontrollers, computers, systems, anything with a serial port. Point to point and multi-point networks are supported.

Figure 3-1: Xbee S2 Module

Features of XBee S2

- 3.3V @ 40mA
- 250kbps Max data rate
- 2mW output (+3dBm)
- 400ft (120m) range
- Built-in antenna
- Fully FCC certified
- 6 10-bit ADC input pins
- 8 digital IO pins
- 128-bit encryption
- Local or over-air configuration
- AT or API command set

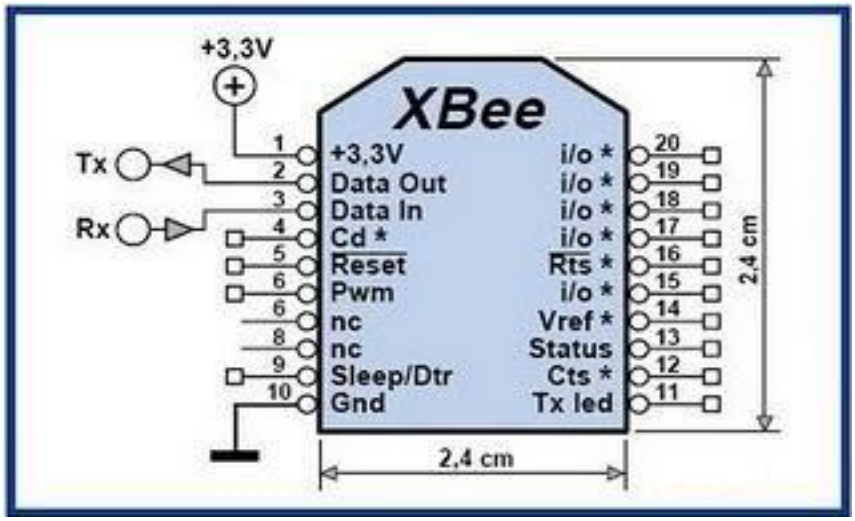


Figure 3-2: Pin configuration of Xbee Module

API vs AT

API system provides the ability to write our own functions that allows us to set packet and buffer size ourselves. The advantage of API is that calculates and appends a CRC code with the transmitted data. At the receiver's end CRC code is regenerated and compared with the transmitter's end code. While this provides us the ability to set our parameters and error detection and correction possibilities, it also adds delay as it requires more processing capabilities. This greatly reduces our efficiency and the aim of this project, which is to achieve communication with minimum delay and without compromising the voice quality.

AT command set provides already made functions with built in parameters. It allows us to utilize these functions according to our requirement while still maintaining a realistic delay.

3.1.2 RASPBERRY PI KIT 2

Raspberry Pi is the controller module being used in this project. It is a small credit card size hardware device that provides efficiency by being portable and easy to configure. In this project, we have used Raspberry Pi Kit 2.

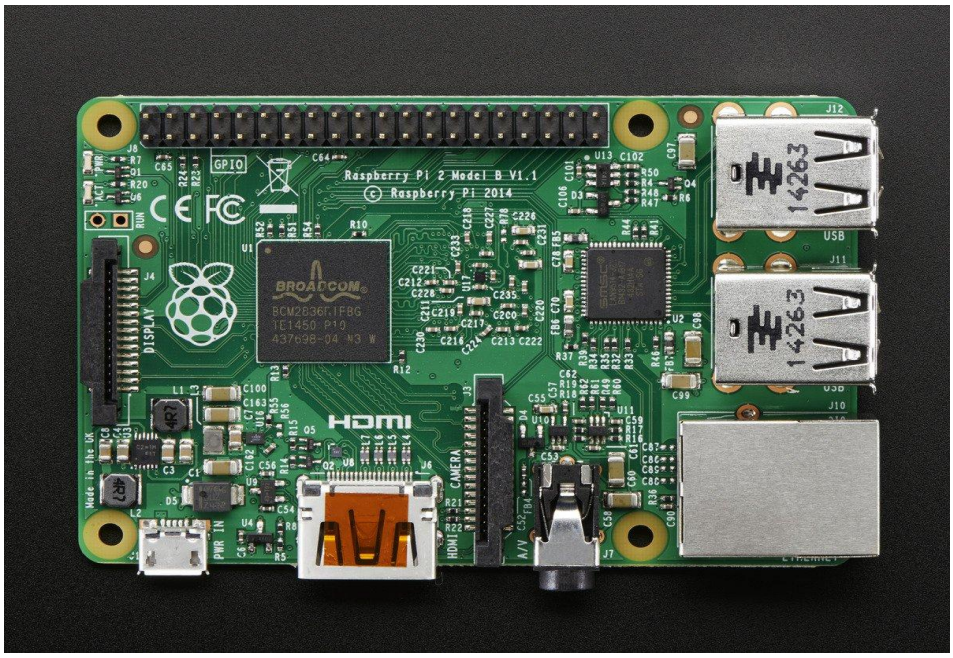


Figure 3-3: Raspberry Pi Kit 2

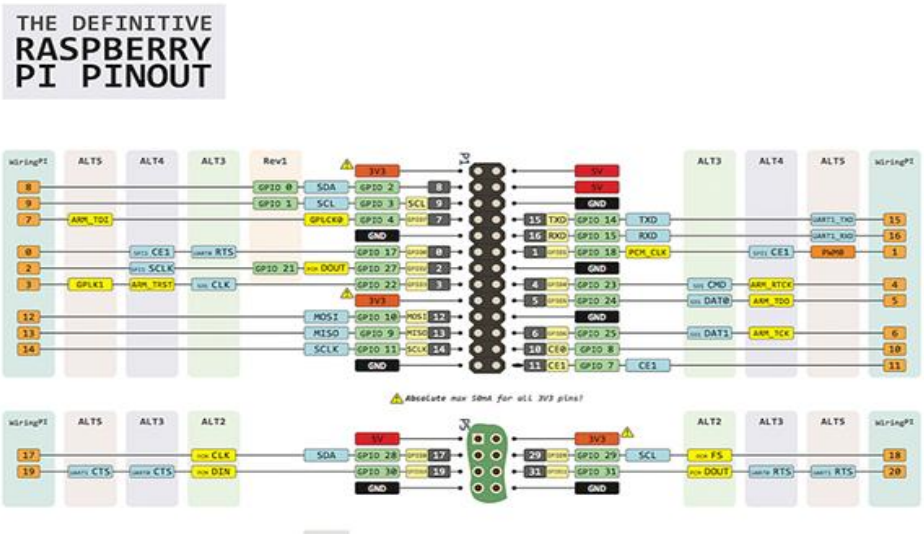
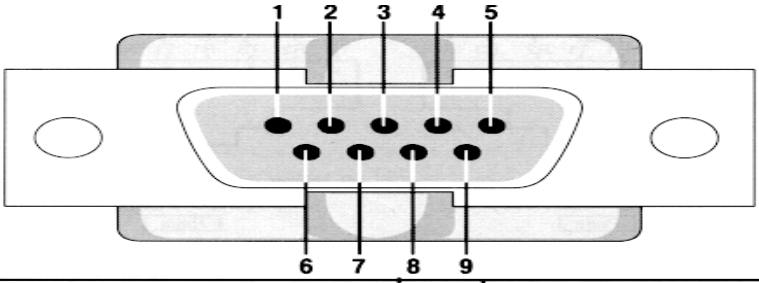


Figure 3-4: Raspberry Pi Pin Layout

3.1.3 RS-232

RS-232 is a standard cable for serial communication transmission of data. It defines the signals connecting between a DTE, for example a computer terminal, and a DCE, for example a modem. The RS-232 standard is used most often in computer serial ports. The standard defines the electrical characteristics and physical of the connectors, and the timing of the signals and pin out of connectors and the meaning of signals.

RS 232 Specifications



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Figure 3-5: Pin Configuration of RS-232

SPECIFICATIONS		RS232	RS423
Mode of Operation		SINGLE -ENDED	SINGLE -ENDED
Total Number of Drivers and Receivers on One Line		1 DRIVER 1 RECVR	1 DRIVER 10 RECVR
Maximum Cable Length		50 FT.	4000 FT.
Maximum Data Rate		20kb/s	100kb/s
Maximum Driver Output Voltage		+/-25V	+/-6V
Driver Output Signal Level (Loaded Min.)	Loaded	+/-5V to +/-15V	+/-3.6V
Driver Output Signal Level (Unloaded Max)	Unloaded	+/-25V	+/-6V
Driver Load Impedance (Ohms)		3k to 7k	>=450
Max. Driver Current in High Z State	Power On	N/A	N/A
Max. Driver Current in High Z State	Power Off	+/-6mA @ +/-2v	+/-100uA
Slew Rate (Max.)		30V/uS	Adjustable
Receiver Input Voltage Range		+/-15V	+/-12V
Receiver Input Sensitivity		+/-3V	+/-200mV
Receiver Input Resistance (Ohms)		3k to 7k	4k min.

Figure 3-6: Table of RS-232 Specifications

3.2 Software

3.2.1 Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) developed by Microsoft. It is used worldwide to develop computer programs for Microsoft Windows, as well as web sites, applications and services. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store and Microsoft Silverlight.

3.2.2 Ubuntu OS

Ubuntu is a Debian-based Linux operating system and distribution for personal computers, smartphones and network servers. It uses Unity as its default user interface. Ubuntu provides a wide variety of libraries and programs that make it easier to implement a program without the need of additional soft wares. Some of the few features of Ubuntu used in this project are

- Minicom
- LibSerial
- PortAudio
- Alsa Sound Libraries

3.3 DESIGN REQUIREMENTS:

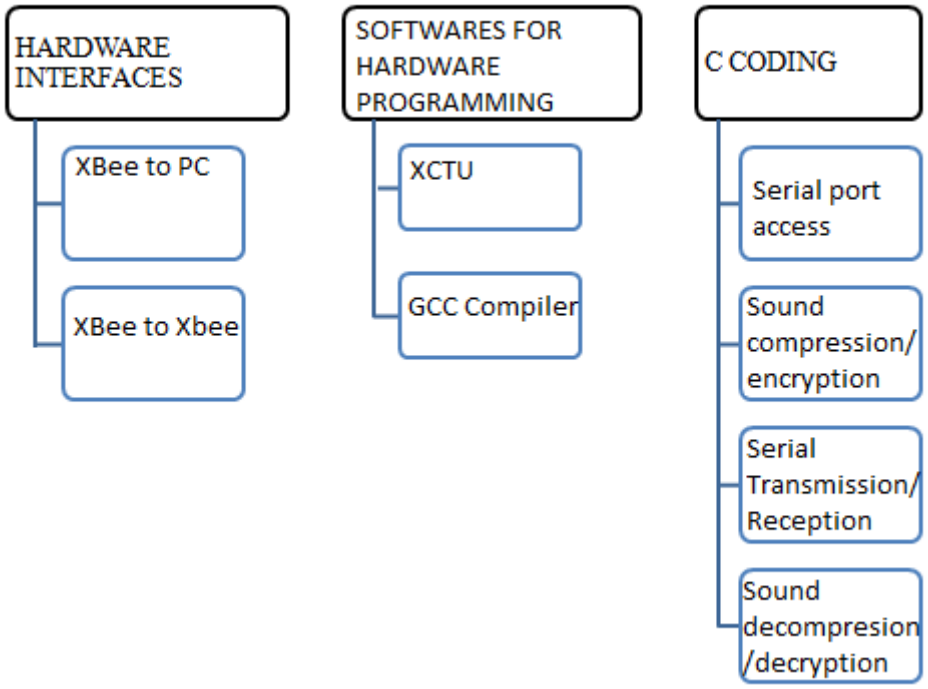


Figure 3-7: Design Requirements

The figure above shows the hardware interfaces, software used and the modules in the project.

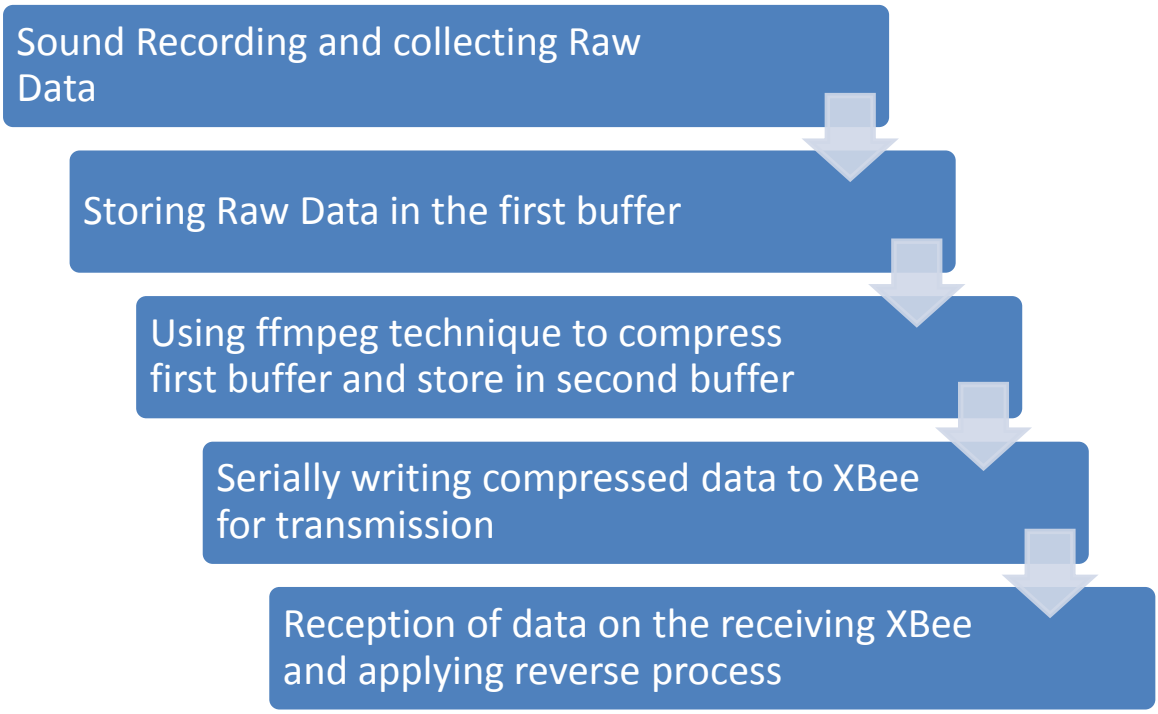


Figure 3-8: Implementation Procedure

The figure above shows our finalized approach towards the project and the implementation procedure that will be followed.

3.4 DESIGN SPECIFICATIONS:

- XBee will be programmed to work according to the required specifications using software XCTU
- Serial port would be accessed and programmed to send and receive data with XBee
- Real time voice packets of 5 seconds would be recorded as raw data and further processed
- Compression and Encryption would be implemented on the voice using a C code in a linux based environment

- The process would repeat itself on every packet of 5 seconds in an infinite loop as to provide continuous communication

3.5 DETAILED DESIGN

3.5.1 Xbee TO PC INTERFACE:

XBee is connected to the Linux based PC using a XBee USB Adapter. Using the XCTU software the XBee is programmed and interfaced with the following specifications:

- Baud Rate
- Bit Rate
- Address defined for the XBee
- Address of the communicating XBee

A C code is written and compiled which helps access the serial port of the PC where the XBee is attached. When the data is sent to the serial port of the PC it is sent to the XBee's transmitting pin ready for transmission.

3.5.2 Xbee TO Xbee INTERFACE:

Using the XBee's software XCTU, they are fed with each other's hardware addresses. Using those addresses both the XBee's search for the other in the air interface inside their range so that a secure communication can be established. This reduces the chances of data transmitting to the wrong XBee. Once paired, both are ready for transmission and reception of data.

3.6 DESIGN DEVELOPMENT

Two different Linux Based PC's are setup each connected to its own XBee. One PC is defined as the transmitter side and the other end is defined as the reception side. On the transmitter PC a C code is compiled and run which records the sound using a mic. Raw data from the mic is collected and stored in a buffer. Each packet records sound of 5 seconds for further processing. The Raw data from the first buffer is compressed using the FFMPEG technique and encoded into mp3 format. After compression the new data is stored in a second buffer. When the second buffer completely fills with the processed data it automatically starts writing it to the serial port. The serial port is connected with the XBee. When it receives data from the PC it starts transmitting it over the air medium to the receiving XBee.

When the raw data is compressed and stored in the second buffer, the first buffer starts to empty in a FIFO order. As to avoid delay in the process, in a simultaneous thread new raw data is collected and stored in the first buffer as it empties. Similarly, when the second buffer empties while sending data to the serial port, more data from the first buffer is processed and compressed, and stored in the second buffer.

At the receiving side, the XBee is constantly receiving data over the air from its paired XBee. The data received is stored inside a buffer. Before it can be used it needs to be decompressed back to its original form. Using the same technique, it is decompressed and transferred to another buffer. When the second buffer fills up, the data is ready and sent to the audio port for playback.

This is an automatic process where packets of 5 seconds are recorded, compressed and transmitted. Simultaneously they are received, decompressed and played back.

Chapter 4

PROJECT ANALYSIS AND EVALUATION

The aim of this project was design a low cost wireless system using commercial off the shelf module, so we could minimize the cost and provide each soldier of the Pakistan Army with a personal hand-set that will allow effective communication.

In order to make the project low cost, we used ZigBee technology, Xbee S2. It has a built-in 128 AES encryption that provides security which no other RF module or WiFi can provide. This allowed us to eliminate any doubt of backdoor attacks. However, in order to achieve voice communication over Xbee S2 we had to compress the voice. Since Xbee is capable of communication over lower data rates, compression was necessary while not compromising on the voice quality. This as mentioned in the previous chapters was done using MPEG compression.

4.1 ANALYSIS AND EVALUATION

In order to test the efficiency of our project, we performed various tests in the 30-meter range, with and without obstacles. It was noted that the project was working and providing effective communication, however, due to the presence of certain obstacles there was a certain delay and interruptions caused during the transmission and reception of voice. In order to minimize the delay and remove the interruptions, we found out it was much more suitable if the sender recorded a voice of lesser duration. This reduced any delay that was occurring and allowed for better and efficient communication.

Chapter 5

FUTURE WORK AND RECOMMENDATION

5.1 OVERVIEW

This project aims at developing technology for low cost wireless module to be used in battle zones for modern warfare. The project will serve as a technology demonstrator for future development of more complex yet compact wireless radio sets using the COTS module. To accommodate the features of a complete radio in the form of a walkie talkie, we are going to develop a prototype of a Zigbee based small radio that would be able to fulfill maximum functions that a soldier requires in battlefield where efficient and good quality communication cannot be compromised upon. We have designed and developed a prototype of a low cost radio using Zigbee Pro S2B as the RF module that is being controlled by a Raspberry Pi board which provides the functions of controller module.

5.2 OBJECTIVES ACHIEVED

The objective of this project was to provide a low-cost radio device and safer way of transmitting data using ZigBee module which is achieved by the in-built AES encryption standard provided in the MAC layer of module. The project was thoroughly tested and it gives the required results satisfactorily. Objectives achieved regarding this project include secure and low cost wireless transmission of voice and coding in C++ for interfacing Raspberry-Pi with different modules of the project. The target specifications of our project are to get working of a complete radio which should later be developed and produced within our own country to ensure security.

5.3 APPLICATION/UTILITY

The application of our project is that its cost effectiveness would cut down the cost being spent on importing radio sets from abroad that are not secure and vulnerable to backdoor

attacks. Most importantly the Zigbee module that we are using provides security within itself and the technology defined by the ZigBee specification is intended to be simpler and less expensive than other wireless personal area networks (WPANs), such as Bluetooth or Wi-Fi.

5.4 LIMITATIONS

Zigbee does have limitations in the area of energy-use restrictions for certification, memory size, processing speed of data and size of bandwidth. As the module is mostly used for wireless transmission of text-based data which is of much smaller size as compared to voice, there might arise certain problems in sending the compressed voice. A high-level compression is required for this task which leads to delays and other related issues.

5.5 FUTURE RECOMMENDATION

Following are the recommendations to be catered for if further work is to be done on the project:

- The project can be further improved by reducing the delay that is occurring. At the moment we cannot reduce the delay more than 10 seconds without compromising on the voice quality.
- The main purpose of the project is to achieve compression without compromising the quality of voice.

- Furthermore, the project could be implemented on microcontrollers which provide much better efficiency and are portable. They could help create a portable model, a proper walkie-talkie that could be used in the battle zones.
- Incorporating handling of large sized files with faster and better speed and by allocating less computer resources.

Chapter 6

REFERENCES

6.1 LIST OF SIMILAR PROJECTS DONE AT NUST

[1] Capt. Aftab Hussain, Capt. Jamal Umair Awan, Capt. M. Asif, Capt. Aamir Riaz, "*Wireless Audio/Video Transmission and Friend or Enemy Recognition",* Military College of Signals, NUST, 2014

[2] Capt. Ahmed Kamal, Capt. Atif Asghar, Capt. Khurram Altaf, "*Voice Communication Over ZIGBEE",* Military College of Signals, NUST, 2013

6.2 BIBLIOGRAPHY

[1] Beginning Ubuntu Linx, 3rd edition by Keir Thomas and Jaime Sicam

[2] Migrating from Windows 7 To Ubuntu: The Ultimate Guide written by Kihara Kimachia

[3] Ubuntu: A Beginner's Guide by MakeUseOf

[4] What is RS232 and Serial Communications? | TALtech www.taltech.com

[5] Serial Communication Using RS-232 www.z80.info

[6] LibSerial 0.6.0rc3 documentation; http://libserial.sourceforge.net/

[7] Minicom Documentation; https://help.ubuntu.com/community/Minicom

6.3 ONLINE HELP

[1] https://help.ubuntu.com/community/VirtualSerialPort

[2] https://en.wikibooks.org/wiki/Using_Ubuntu_Linux

[3] http://www.siongboon.com/projects/ 2006 – 03 -06_serial_communication /#RS48%20Interfacing

[4] http://extremeelectronics.co.in/avr-tutorials/rs232-communication-the-basics/

[5] https://www.sparkfun.com/products/10414

[6] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[7] https://learn.sparkfun.com/tutorials/exploring-xbees-and-xctu

[8] http://tutorial.cytron.com.my/2012/03/08/xbee-series-2-point-to-point-communication/

APPENDEX A

CODES

APPENDEX A1

SERIAL PORT CODE

```
#include "serial.h"

//#define _TX_SIDE_
//#define _RX_SIDE_

int set_interface_attribs(int fd, int speed, int parity) {
        struct termios tty;
        memset(&tty, 0, sizeof tty);
        if (tcgetattr(fd, &tty) != 0)
        {
                perror("error from tcgetattr");
                return -1;
        }

        cfsetospeed(&tty, speed);
        cfsetispeed(&tty, speed);

        tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;     // 8-bit chars
        // disable IGNBRK for mismatched speed tests; otherwise receive break
        // as \000 chars
        tty.c_iflag &= ~IGNBRK;         // disable break processing
        tty.c_lflag = 0;                // no signaling chars, no echo,
                                                                        // no canonical
processing
        tty.c_oflag = 0;                // no remapping, no delays
        tty.c_cc[VMIN] = 0;                     // read doesn't block
        tty.c_cc[VTIME] = 0.5;          // 0.5 seconds read timeout

        tty.c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/xoff ctrl
        tty.c_cflag |= (CLOCAL | CREAD);// ignore modem controls,
                                                                        // enable reading
        tty.c_cflag &= ~(PARENB | PARODD);      // shut off parity
        tty.c_cflag |= parity;
        tty.c_cflag &= ~CSTOPB;
        tty.c_cflag &= ~CRTSCTS;

        if (tcsetattr(fd, TCSANOW, &tty) != 0) {
                perror("error from tcsetattr");
                return -1;
        }
        return 0;
}

void set_blocking(int fd, int should_block) {
        struct termios tty;
        memset(&tty, 0, sizeof tty);
```

```
        if (tcgetattr(fd, &tty) != 0) {
                perror("error from tggetattr");
                return;
        }

        tty.c_cc[VMIN] = should_block ? 128 : 0;
  //tty.c_cc[VMIN] = should_block ? 1 : 0;
        tty.c_cc[VTIME] = 5;                                    // 0.5 seconds read
timeout

        if (tcsetattr(fd, TCSANOW, &tty) != 0)
                perror("error setting term attributes");
}

#ifdef _TX_SIDE_
char *portname = "/dev/ttyUSB1";

int main() {
        int i = 0;
        int fd = open(portname, O_RDWR | O_NOCTTY | O_SYNC);
        if (fd < 0) {
                perror("error opening port");
                return -1;
        }

        set_interface_attribs(fd, B115200, 0);                  // set speed to 115,200 bps,
8n1 (no parity)
        set_blocking(fd, 1);
                // set blocking

        int file_fd = open("output.mp3", O_RDONLY);
        if (fd < 0) {
                perror ("error opening file");
                return -1;
        }

        int length = lseek(file_fd, 0, SEEK_END) + 1;
        printf("file length: %d\n", length/128);
        lseek(file_fd, 0, SEEK_SET);
        char blocks = length/128;
        if (length % 128)
                blocks ++;

        char buf[128];
        int n = 0;
```

```
	write(fd, &blocks, 1);
	usleep(500000);

	while (1) {
		usleep(50000);
		n = read(file_fd, buf, sizeof buf);			// read up to 128
characters
		if (n == 0) {
			printf("end of file!\n");
			break;
		}
		printf ("read %d bytes\n", n);
		n = write(fd, buf, n);
		if (n != sizeof buf)
			printf("short write: %d\n", n);
	}

	close(fd);
	close(file_fd);
	return 0;
}
#endif

#ifdef _RX_SIDE_
char *portname = "/dev/ttyUSB0";

int main() {
	int i = 0;
	int fd = open(portname, O_RDWR | O_NOCTTY | O_SYNC);
	if (fd < 0) {
		perror("error opening port");
		return -1;
	}

	set_interface_attribs(fd, B115200, 0);			// set speed to 115,200 bps,
8n1 (no parity)
	set_blocking(fd, 1);
		// set blocking

	int file_fd = open("input.mp3", O_RDWR | O_CREAT, 0644);
	if (fd < 0) {
		perror ("error opening file");
		return -1;
	}

	char buf[128];
```

```
	int n = 0;
	char blocks;

	tcflush(fd,TCIOFLUSH);
	while (1) {
		n = read(fd, &blocks, 1);
		printf("blocks: %02X\n", blocks);
		while (blocks) {
			blocks--;
			n = read(fd, buf, sizeof buf);                    // read
up to 64 characters
			if (n == 0) {
				printf("end of file!\n");
				break;
			}
			printf("read %d bytes\n", n);
			n = write(file_fd, buf, sizeof buf);
			if (n != sizeof buf)
				printf("short write: %d\n", n);
		}
		break;
	}
	close(fd);
	close(file_fd);
	return 0;
}
#endif
```

APPENDEX A2

RECEIVING AND PLAYBACK CODE

```
/* Use the newer ALSA API */
#define ALSA_PCM_NEW_HW_PARAMS_API

#include <alsa/asoundlib.h>
#include <pthread.h>
#include <signal.h>
#include "serial.h"

// LOCAL DATA
static unsigned char keep_running = 1;
static char *portname = "/dev/ttyUSB0";
static unsigned char playback = 0;              // stream buffer has atleast one sec data to play
int size;
int playback_fd;


/* signal handler to stop program */
void signal_callback_handler(int signum) {
  printf("Aborted by signal Interrupt...\n");
  keep_running = 0;
  // Terminate program
  //exit(signum);
}

void *Playback() {
  int rc;
  snd_pcm_t *handle;
  snd_pcm_hw_params_t *params;
  unsigned int val;
  int dir;
  snd_pcm_uframes_t frames;
          char *buffer;

  /* Open PCM device for playback. */
  rc = snd_pcm_open(&handle, "default", SND_PCM_STREAM_PLAYBACK, 0);
  if (rc < 0) {
    fprintf(stderr, "unable to open pcm device: %s\n",
          snd_strerror(rc));
    exit(1);
  }

  /* Allocate a hardware parameters object. */
  snd_pcm_hw_params_alloca(&params);
  /* Fill it in with default values. */
  snd_pcm_hw_params_any(handle, params);
```

```
/* Set the desired hardware parameters. */
/* Interleaved mode */
snd_pcm_hw_params_set_access(handle, params,
                SND_PCM_ACCESS_RW_INTERLEAVED);
/* Unsigned 8-bit format */
snd_pcm_hw_params_set_format(handle, params,
                      SND_PCM_FORMAT_U8);
/* Single channels (mono) */
snd_pcm_hw_params_set_channels(handle, params, 1);
/* 8000 bits/second sampling rate */
val = 8000;
snd_pcm_hw_params_set_rate_near(handle, params,
                        &val, &dir);
/* Set period size to 32 frames. */
frames = 32;
snd_pcm_hw_params_set_period_size_near(handle,
                      params, &frames, &dir);
/* Write the parameters to the driver */
rc = snd_pcm_hw_params(handle, params);
if (rc < 0) {
  fprintf(stderr, "unable to set hw parameters: %s\n",
        snd_strerror(rc));
  exit(1);
}
/* Use a buffer large enough to hold one period */
snd_pcm_hw_params_get_period_size(params, &frames,
                        &dir);
size = frames * 1; /* 1 bytes/sample, 1 channel */
        buffer = (char *) malloc(size);

while (keep_running) {

                if (playback) {
                        // read from raw stream file
                        rc = read(playback_fd, buffer, size);
        if (rc == 0) {
        fprintf(stderr, "end of file on input\n");
                                close(playback_fd);
        playback = 0;
                                continue;
        } else if (rc != size) {
        fprintf(stderr, "short read: read %d bytes\n", rc);
        }

   // write buffer to alsa device
   rc = snd_pcm_writei(handle, buffer, frames);
```

```
    if (rc == -EPIPE) {
      /* EPIPE means underrun */
      fprintf(stderr, "underrun occurred\n");
      snd_pcm_prepare(handle);
      continue;
    }
    else if (rc < 0) {
      fprintf(stderr, "error from writei: %s\n",
           snd_strerror(rc));
      continue;
    }
    else if (rc != (int)frames) {
      fprintf(stderr, "short write, write %d frames\n", rc);
    }
   }
                else
                        usleep(100);
  }

          free(buffer);

  snd_pcm_drain(handle);
  snd_pcm_close(handle);
}

int main() {
  pthread_t playback_th;
  int rc;
          char blocks;

          int stream = 0;
          int stream_fd;
          char *buffer;
          char ffmpeg_cmd[500];

          system ("rm -f input.mp3 output.raw > /dev/null 2>&1");

  /* open serial port */
  int fd = open(portname, O_RDWR | O_NOCTTY | O_SYNC);
  if (fd < 0) {
            perror("error opening port");
            return -1;
  };
  /* set speed to 115,200 bps, 8n1 (no parity) */
  set_interface_attribs(fd, B115200, 0);
  /* set blocking */
```

```
 set_blocking(fd, 1);

 // start playback thread
 rc = pthread_create(&playback_th, NULL, Playback, NULL);
 if(rc) {
   fprintf(stderr, "Error - pthread_create() return code: %d\n", rc);
   exit(EXIT_FAILURE);
 }

 // Register signal and signal handler
 signal(SIGINT, signal_callback_handler);

 // wait for playback stream to prepare
 usleep(500000);
 printf("stream buffer size: %d\n", size);
        buffer = malloc(size);

        // open mp3 file for streaming
        stream_fd = open("input.mp3", O_WRONLY | O_CREAT, 0644);
        if(-1 == stream_fd) {
                perror("stream file open failed");
                exit(EXIT_FAILURE);
        }
        // flush serial
        tcflush(fd, TCIOFLUSH);
 /* read from serial port, write to mp3 file */
 while (keep_running) {
                rc = read(fd, &blocks, 1);
                printf("file blocks: %02X\n", blocks);
                if(blocks == 0x00)
                        continue;
                while(blocks > 0) {
                        blocks--;
        rc = read(fd, buffer, size);
        if (rc == 0) {
        fprintf(stderr, "end of file on input\n");
                                goto file_end;
        } else if (rc != size) {
        fprintf(stderr, "short read: read %d bytes\n", rc);
        }
                        // write to mp3 file
                        rc = write(stream_fd, buffer, size);
                        if (rc != size)
                fprintf(stderr, "short write: wrote %d bytes\n", rc);
                }
file_end:
```

```
                close(stream_fd);
                // convert mp3 to raw pcm for playback
                system("avconv -y -i input.mp3 -f u8 -ar 8000 -ac 1 output.raw > /dev/null
2>&1");
                // open playback file and signal playback thread
                playback_fd = open("output.raw", O_RDONLY);
                if(-1 == playback_fd) {
                        perror("playback file open failed");
                        exit(EXIT_FAILURE);
                }
                playback = 1;
                // reopen mp3 file for streaming
                stream_fd = open("input.mp3", O_WRONLY | O_CREAT, 0644);
                if(-1 == stream_fd) {
                        perror("stream file open failed");
                        exit(EXIT_FAILURE);
                }
                blocks = 0;
                // flush serial
                //tcflush(fd, TCIOFLUSH);
                continue;
  }

        printf("waiting for playback thread to exit\n");
  pthread_join(playback_th, NULL);

        free(buffer);
  close(fd);

  return 0;
}
```

APPENDEX A3

RECORDING AND SENDING CODE

```
/* Use the newer ALSA API */
#define ALSA_PCM_NEW_HW_PARAMS_API

#include <alsa/asoundlib.h>
#include "serial.h"
#include <signal.h>
#include <pthread.h>

// ffmpeg -y -f u8 -ar 8000 -ac 1 -i audio_1.raw -f mp3 output.mp3
// ffmpeg -y -i output.mp3 -f u8 -ar 8000 -ac 1 output.raw


static char *portname = "/dev/ttyUSB0";
static unsigned char keep_running = 1;
static int stream = 0;
static int size;

/* signal handler to stop program */
void signal_callback_handler(int signum) {
  printf("Aborted by signal Interrupt...\n");
  keep_running = 0;
}

void *stream_file() {
        int rc;
        char blocks;
        int length;
        int stream_fd;
        char *strm_buffer;

        /* open serial port */
  int fd = open(portname, O_RDWR | O_NOCTTY | O_SYNC);
  if (fd < 0) {
          perror("error opening port");
          exit(EXIT_FAILURE);
  }
  /* set speed to 115,200 bps, 8n1 (no parity) */
  set_interface_attribs(fd, B115200, 0);
  /* set blocking */
  set_blocking(fd, 1);

        strm_buffer = (char *) malloc(size);

        while (keep_running) {
                if (stream) {
                        // open mp3 file for streaming
```

```
                        stream_fd = open("output.mp3", O_RDONLY);
                        if(-1 == stream_fd) {
                                perror("stream file open failed");
                                exit(EXIT_FAILURE);
                        }
                        // get file size for serial transfer
                        length = lseek(stream_fd, 0, SEEK_END) + 1;
                        lseek(stream_fd, 0, SEEK_SET);
                        char blocks = length/size;
                        if(length % size)
                                blocks ++;
                        printf("file blocks: %02X\n", blocks);

                        // write number on serial
                        write(fd, &blocks, 1);
                        usleep(50000);

                        while(blocks) {
                                blocks--;
                                rc = read(stream_fd, strm_buffer, size);
                if (rc == 0) {
                fprintf(stderr, "end of file on input\n");
                                        break;
                } else if (rc != size) {
        fprintf(stderr, "short read: read %d bytes\n", rc);
                }
                                // write stream data
                                rc = write(fd, strm_buffer, size);
                if (rc != size)
                fprintf(stderr, "short write: wrote %d bytes\n", rc);
                                // wait for packet tx to complete
                                usleep(50000);
                        }
                        close(stream_fd);
                        stream = 0;

                }
                else
                        usleep(100);
        }
        free(strm_buffer);
        close(fd);
}

int main() {
```

```
        pthread_t stream_th;

        long loops;
        long run;
int rc;
snd_pcm_t *handle;
snd_pcm_hw_params_t *params;
unsigned int val;
int dir;
snd_pcm_uframes_t frames;
char *buffer;

        int file_fds[2];
        int fd_index;
        char filename[50];

        char ffmpeg_cmd[500];

        system ("rm -f audio_* output.mp3 > /dev/null 2>&1");

/* Open PCM device for recording (capture). */
rc = snd_pcm_open(&handle, "default", SND_PCM_STREAM_CAPTURE, 0);
if (rc < 0) {
  fprintf(stderr, "unable to open pcm device: %s\n",
        snd_strerror(rc));
  exit(1);
}

/* Allocate a hardware parameters object. */
snd_pcm_hw_params_alloca(&params);
/* Fill it in with default values. */
snd_pcm_hw_params_any(handle, params);
/* Set the desired hardware parameters. */
/* Interleaved mode */
snd_pcm_hw_params_set_access(handle, params,
                SND_PCM_ACCESS_RW_INTERLEAVED);
/* Unsigned 8-bit format */
snd_pcm_hw_params_set_format(handle, params,
                        SND_PCM_FORMAT_U8);
/* Single channels (mono) */
snd_pcm_hw_params_set_channels(handle, params, 1);
/* 8000 bits/second sampling rate */
val = 8000;
snd_pcm_hw_params_set_rate_near(handle, params,
                           &val, &dir);
/* Set period size to 32 frames. */
```

```
frames = 32;
snd_pcm_hw_params_set_period_size_near(handle,
                      params, &frames, &dir);
/* Write the parameters to the driver */
rc = snd_pcm_hw_params(handle, params);
if (rc < 0) {
  fprintf(stderr, "unable to set hw parameters: %s\n",
        snd_strerror(rc));
  exit(1);
}
/* Use a buffer large enough to hold one period */
snd_pcm_hw_params_get_period_size(params,
                           &frames, &dir);
size = frames * 1 * 1; /* 1 bytes/sample, 1 channel */
buffer = (char *) malloc(size);

/* We want to loop for 5 seconds */
snd_pcm_hw_params_get_period_time(params,
                          &val, &dir);
/* 5 seconds in microseconds divided by
 * period time for recording*/
run = loops = 5000000 / val;
printf("stream buffer size: %d\n", size);

        // start streaming thread
rc = pthread_create(&stream_th, NULL, stream_file, NULL);
if(rc) {
  fprintf(stderr, "Error - pthread_create() return code: %d\n", rc);
  exit(EXIT_FAILURE);
}

        // open files for recording
        fd_index = 0;
        snprintf(filename, sizeof(filename), "audio_%d.raw", fd_index);
        file_fds[fd_index] = open(filename, O_RDWR | O_CREAT, 0644);
        if(-1 == file_fds[1]) {
                perror("file open failed");
                exit(EXIT_FAILURE);
        }

// Register signal and signal handler
signal(SIGINT, signal_callback_handler);

/* read from alsa stream for 5 sec,
                convert to low bitrate & write to
                serial port for record */
```

```
 while (keep_running) {
                    // read from alsa record device
   rc = snd_pcm_readi(handle, buffer, frames);
   if (rc == -EPIPE) {
     // EPIPE means overrun
     fprintf(stderr, "overrun occurred\n");
     snd_pcm_prepare(handle);
   } else if (rc < 0) {
     fprintf(stderr, "error from read: %s\n",
           snd_strerror(rc));
   } else if (rc != (int)frames) {
     fprintf(stderr, "short read, read %d frames\n", rc);
   }
                    // write raw stream to file
   rc = write(file_fds[fd_index], buffer, size);
   if (rc != size)
     fprintf(stderr, "short write: wrote %d bytes\n", rc);

                    // loop five sec
                    if(run) {
                              run --;
                    }
                    // 5 sec raw file complete
                    else {
                              run = loops;
                              close(file_fds[fd_index]);
                              // convert file to mp3 for transfer
                              snprintf(ffmpeg_cmd, sizeof(ffmpeg_cmd), "avconv -y -f u8 -ar
8000 -ac 1 -i %s -f mp3 output.mp3 > /dev/null 2>&1", filename);
                              system(ffmpeg_cmd);

                              // open secondary file for raw buffering
                              fd_index = (fd_index + 1) % 2;
                              snprintf(filename, sizeof(filename), "audio_%d.raw", fd_index);
                              file_fds[fd_index] = open(filename, O_RDWR | O_CREAT,
0644);
                              if(-1 == file_fds[fd_index]) {
                                        perror("file open failed");
                                        exit(EXIT_FAILURE);
                              }
                              stream = 1;
                    }
 }

          printf("waiting for stream thread to exit\n");
 pthread_join(stream_th, NULL);
```

```
            close(file_fds[fd_index]);

    snd_pcm_drain(handle);
    snd_pcm_close(handle);

    free(buffer);

    return 0;
}
```